TP 3: Fonction Inverse

1 Fonctions d'une variable

Le problème que nous allons traiter dans cet exercice est assez générale. Supposons qu'on a une fonction `f :: Integer -> Integer`. On aimerait avoir une opération qui calcule l'inverse de cette fonction. Par exemple, on aimerait pouvoir écrire `(invert f) (f 5)` et que cela donne `5`, quelle que soit la fonction `f`. En termes mathématiques, l'opération `invert f` correspondrait à la fonction f^{-1}. Avant d'avancer plus loin, prenez un moment pour penser quel devrait être le type de la fonction `invert`.

Dans le cas générale ce problème est extrêmement difficile à résoudre[1]. Donc, pour simplifier les choses, dans cet exercice on va se concentrer sur des fonctions `f` qui satisfont les propriétés suivantes :

1. `f::Integer -> Integer`
2. On supposera que les arguments donnés sont des entiers positifs, et que $\forall$ `x>0` on a `f x > 0`
3. On supposera que `f` est croissante, c'est-à-dire, $\forall$ `x` on a `f (x+1) >= f x`.

Par exemple, trois fonctions qui satisfont ces conditions sont :

```
f1 x = x^3 + 3*x^2 + 7
f2 x = x^2 + 7*x + 250
f3 x = (x`div`100)^2+5
```

On vous demande de programmer (au moins) les fonctions suivantes :

1. `invertBrute :: (Integer -> Integer) -> Integer -> [Integer]`
2. `invertBin :: (Integer -> Integer) -> Integer -> [Integer]`

Les deux fonctions devraient avoir le même comportement : étant donné une fonction `f` et un entier `t`, l'expression `invert f t` devra retourner une liste qui contient l'entier **minimum** `x` tel que `f x == t`. Si aucun tel entier n'existe, la fonction devra retourner la liste vide.

Exemples d'utilisation :

```
*Main> invertBrute f1 7
[0]
*Main> invertBrute f1 27
[2]
*Main> invertBrute f1 6
[]
*Main> invertBrute f2 (f2 123)
[123]
*Main> invertBrute f3 (f3 123)
[100]
```

Dans tous les cas le résultat devrait être le même, qu'on utilise `invertBrute` ou `invertBin`. Notez que dans le dernier exemple on returne `[100]` et pas `[123]`, car 100 est l'entier `x` **minimum** tel que `f3 x == f3 123`.

[1] Et c'est une bonne nouvelle que ce problème est difficile en général ! Par exemple, imaginez que `f` pourrait être une fonction cryptographique qui retourne une version crypté de son argument. Évidemment, ça ne devrait pas être facile de l'inverser !

Considérations Algorithmiques

Vous vous posez sans doute la question, si les deux fonctions demandées retournent le même résultat, pourquoi programmer les deux. Comme indiqué par leurs noms, la différence entre les deux fonctions est algorithmique : `invertBrute` utilisera un algorithme de recherche brute-force, alors que `invertBin` utilisera l'algorithme de recherche binaire (`https://fr.wikipedia.org/wiki/Recherche_dichotomique`).

Plus précisément, pour `invertBrute` vous pouvez utiliser une implémentation très simple qui essaie toutes les valeurs possible de `x` jusqu'au moment où on trouve `f x == t` où qu'on a `f x > t` (dans ce cas, ce n'est pas la peine de continuer la recherche, vu que la fonction est non-décroissante). Vous utiliserez sans doute une expression du style `x<-[0..]` qui traverse tous les entiers positifs.

Pour `invertBin` vous utiliserez l'algorithme (récursif) classique qui, étant donné un intervalle $[a, b]$ le découpe en deux parties de la même taille et décide où on pourrait trouver l'élément qui nous intéresse selon la valeur que `f` donne pour $\frac{a+b}{2}$. Puisque l'intervalle initial ne peut pas être $[0, \infty]$, vous devrez trouver une façon d'initialiser cet intervalle. Pour cela, il vous faudra calculer rapidement un `b` tel que `f b >= t`.

Quelques commentaires par rapport aux types

Vous avez constaté que le type qu'on vous demande est `(Integer -> Integer) -> Integer -> [Integer]`. L'opération `invert` est une fonction d'ordre supérieur, donc c'est normal que le type du premier argument soit `(Integer->Integer)`. Mais pourquoi est-ce que cette fonction retourne `[Integer]` et pas `Integer` ?

La raison principale pour laquelle on a choisi ce type est que c'est possible que quelqu'un nous demande de calculer par exemple `(invert f1) 6`, alors que `f1` ne retourne jamais `6`. À la place de retourner une erreur qui arrêtera l'exécution de notre programme, c'est mieux de retourner une valeur qui signifie l'impossibilité de trouver un résultat, dans notre cas une liste vide. Ceci n'est pas la solution la plus élégante et on verra plus tard que Haskell nous offre un type spécialisé pour cette utilisation, le type `Maybe`.

Une autre complication de ce choix est que ce n'est pas possible d'enchaîner facilement l'application d'une fonction inverse avec d'autres fonctions. Par exemple, l'expression mathématique $f^{-1}(f(f^{-1}(f(5))))$ est correcte est donne 5, alors que `f (invertBrute f 5)` donne une erreur de type. Haskell nous offre une solution pour ce problème qu'on va voir plus tard quand on parlera des `Monad`s.

2 Fonctions de Deux Variables

Pour la deuxième partie, vous devez résoudre le même problème mais pour des fonctions de deux variables. Pour simplifier votre tâche vous pouvez supposer que les fonctions sont maintenant **strictement croissantes**, c'est-à-dire, $\forall$ `x,y` on a `f (x+1) y > f x y` et `f x (y+1) > f x y`. On vous demande de programmer les fonctions suivantes :

1. `invertBruteTwo::(Integer->Integer->Integer)->Integer->[(Integer,Integer)]`
2. `invertBinTwo :: (Integer->Integer->Integer)->Integer->[(Integer,Integer)]`

Ces deux fonctions prennent comme arguments une fonction `f` et un entier `t` et retournent une liste qui contient une paire `(x,y)` telle que `f x y == t` (ou une liste vide si aucune telle paire n'existe).

Exemples d'utilisation :

```
*Main> g1 x y = x^2 + y^3 + x*y + 2
*Main> g2 x y = 15*x + y^2 + 3*x*y + 7
*Main> invertBruteTwo g1 (g1 23 43)
[(23,43)]
*Main> invertBruteTwo g2 (g2 100 100)
[(24,170)]
```

Notez que dans le dernier exemple la valeur retournée est acceptable, car `g2 100 100 == g2 24 170`.

Considérations Algorithmiques

Comme dans la partie précédente, vous devez programmer une fonction qui fait une recherche brute-force et une fonction qui utilise la recherche binaire. Dans les deux cas vous pouvez utiliser le fait que pour calculer `invertBruteTwo f t` il suffit de considérer les paires (x, y) telles que $x, y \in [0, t]$, car la fonction `f` est supposée strictement croissante et par conséquent `f (t+1) 0>t`.

Pour les deux parties de ce TP il sera utile de mesurer le temps d'exécution de vos fonctions. Si vous avez tout programmé correctement, les fonctions qui utilisent la recherche binaire devront être clairement plus performantes quand l'espace de recherche devient plus important. Dans `ghci` vous pouvez activer la fonctionnalité qui mesure le temps d'exécution avec la commande `:set +s`.

Exemples :

```
*Main> invertBrute f1 (f1 (10^6))
[1000000]
(3.37 secs, 1,624,081,632 bytes)
*Main> invertBrute f1 (f1 (10^7))
[10000000]
(34.39 secs, 17,365,014,416 bytes)
*Main> invertBin f1 (f1 (10^7))
[10000000]
(0.02 secs, 283,464 bytes)
*Main> invertBin f1 (f1 (10^10))
[10000000000]
(0.01 secs, 390,368 bytes)
*Main> invertBin f1 (f1 (10^15))
[1000000000000000]
(0.01 secs, 554,208 bytes)
*Main> invertBruteTwo g1 (g1 1000 1000)
[(1000,1000)]
(3.50 secs, 1,680,597,032 bytes)
*Main> invertBruteTwo g1 (g1 2000 2000)
[(2000,2000)]
(13.91 secs, 6,705,112,080 bytes)
*Main> invertBinTwo  g1 (g1 2000 2000)
[(2000,2000)]
(0.79 secs, 346,595,784 bytes)
*Main> invertBinTwo  g1 (g1 5000 5000)
[(5000,5000)]
(2.13 secs, 951,345,616 bytes)
```

Les temps d'exécution donnés ci-dessus sont indicatifs (cela dépend aussi de la puissance de votre ordinateur). Ce qui est important ici est d'observer la différence entre les deux implémentations.

Pour ceux/celles qui s'intéressent à la complexité algorithmique, les complexités souhaitées sont : $O(\log n)$ pour rechercher un espace de taille n pour `invertBin` ; $O(n \log n)$ pour rechercher un espace de taille n^2 pour `invertBinTwo`. Les complexités respectives pour la recherche brute force sont $O(n)$ et $O(n^2)$. Donc, c'est logique que dans tous les cas la recherche binaire soit clairement plus performante, et que cette différence soit plus nette pour les fonctions d'une variable.